Testing REST API Servers

Building Confidence in Your Web Services

🎯 Learning Objectives

By the end of this lesson, you will understand:

- **Why** we need to test REST APIs
- **How** to test API endpoints using **JavaScript** and **CURL**
- **What** makes a good API test
- **When** to run different types of tests

Why and How to test REST APIs?

🤔 Why Test REST APIs?

Reliability 🛡️

- Ensure your API works as expected
- Catch bugs before users do
- Maintain service quality

Confidence 💪

- Safe to make changes
- Deploy with assurance
- Reduce production issues

Documentation 📚

- Tests serve as living examples
- Show how the API should be used
- Prove API functionality

Communication 🗣️

- Clear expectations
- Team understanding
- Client integration

🔍 How - Two Types of API Tests

1. Connection Tests 🌐 "Is the server alive?"

- Server responds to requests
- Returns valid HTTP status codes
- Basic connectivity verification

2. Functional Tests ⚙️ "Does it work correctly?"

- Data validation
- Business logic verification
- Expected behavior confirmation

🛠️ Testing Tools: JavaScript vs CURL

JavaScript/Fetch 🌐

- Web-based testing
- Visual interfaces
- Great for beginners
- Interactive learning

CURL 💻

- Command-line tool
- Universal availability
- Scriptable automation
- Professional standard

Both test the same API - different approaches!

📊 Our Student Management API

```
GET    /students      → Get all students
GET    /students/1    → Get student by ID
POST   /students      → Create new student
PUT    /students/1    → Update student
DELETE /students/1    → Delete student
```

Student Data Model

```
{
  "id": 1,
  "name": "Alice Johnson",
  "email": "alice@university.edu",
  "major": "Computer Science",
  "year": 3
}
```

Shell Programming

- On Linux and macOS, shell scripts are the most common choice for REST API testing.
- On Windows, PowerShell is the modern equivalent, though many developers still prefer Bash scripts (via Git Bash or WSL) for cross-platform compatibility.
- In this section, we focus mainly on shell scripts.

test_get_all.sh

- A variable in a shell script is assigned with `=`, but notice that there is no space between the operator.
- The variable is used with $ prepended.

```
API_BASE_URL="http://localhost:8000"
VERBOSE=false

# Colors for output
RED='\033[0;31m'
GREEN='\033[0;32m'
YELLOW='\033[1;33m'
BLUE='\033[0;34m'
MAGENTA='\033[0;35m'
CYAN='\033[0;36m'
NC='\033[0m' # No Color

echo -e "${GREEN}✅ $test_name: $message${NC}"
```

Function

- Functions are defined as follows.
- The arguments are assigned as $1, $2, and so on.

```
print_result() {
    local test_name= "$1"
    local success= "$2"
    local message= "$3"
    local data= "$4"
```

If Statement

- If/else statement is used as follows.
- The conditional expression is inside the `[[ ... ]];` block.
- We can add the `else` block.

```
    if [[ "$success" == "true" ]]; then
        echo -e "${GREEN}✅ $test_name: $message${NC}"
    else
        echo -e "${RED}❌ $test_name: $message${NC}"
    fi
}
```

echo & function call

- To print out the result, we use `echo`.
- To call the shell function, we use the name of the function followed by matching arguments.

```
    if [[ "$VERBOSE" == "true" && -n "$data" ]]; then
        echo -e "${CYAN}   Response: $data${NC}"
    fi

...

print_result('name',' success', 'message', 'data');
```

make_curl_request

- The curl command needs multiple arguments.
- We can simplify it using a shell function.

```
make_curl_request() {
    local url= "$1"
    local method= "$2"
    local data= "$3"
    local content_type=" application/json"

    if [[ -n "$data" ]]; then
        curl -s -w "\n%{http_code}" -X "$method" \
             -H "Content-Type: $content_type" \
             -d "$data" \
             "$url"
    else
        curl -s -w "\n%{http_code}" -X "$method" "$url"
    fi
}
```

- In this example, we call the `curl` command with various arguments.
- Using the if statement, we separate the case when the `$data` is given or not.
- This is the case when data is not given.
    - `-s' silences progress and error messages.
    - `-w "\n%{http_code}"` appends the HTTP status on a new line after the response body.
    - `-X "$method``` specifies the HTTP method to use (GET, POST, etc).

- We need "\n%{http_code}" option to use the HTTP status to check if the test is success or not.
    - We get the HTTP status to get the last line (tail -n1).

```
response=$(curl -s -w "%{http_code}" \
    http://localhost:8000)
http_code=$(echo "$response" | tail -n1)

if [ "$http_code" = "200" ]; then
    echo "✅ Server running!"
else
    echo "❌ Connection failed"
fi
```

Getting the JSON data except for the HTTP Code

- We need to get the JSON data except for the last line.
- However, this code does not work for FreeBSD-based UNIX systems (such as Mac).

```
local json_data=$(echo "$response" | head -n -1)
```

- Instead, we should use this command to get the lines except for the last one (sed $d).

```
extract_json_data() {
    local response= "$1"
    echo "$response" | sed '$d'
}
local json_data=$(extract_json_data "$response")
```

Get the HTTP code and JSON data

- So, this is the pattern to get the HTTP code and JSON data from the REST API server to explain the idea in the examples.

```
http_code=$(echo "$response" | tail -n1)
json_data=$(echo "$response" | head -n -1)
if [ "$http_code" = "201" ]; then
```

Use the Grep command to extract information

- `grep` searches for patterns in a stream or file — here, it's looking for the exact text `"success":true`
  - The `-q` flag stands for quiet (or silent):
  - It suppresses output — nothing is printed to the terminal.
  - Instead, it uses the exit code to indicate if a match was found.

```
if echo "$response" | grep -q '"success":true'; then
    echo "✅ Success field found"
fi
```

```
{
  "id": 123,
  "name": "Alice",
  "success": true
}
```

- We need to extract the `123` from this JSON string.
- In this case, we can use grep.

```
student_id=$(echo "$json_data" | \
    grep -o '"id":[0-9]*' | \
    grep -o '[0-9]*')
```

- `grep -o '"id":0-9*" Searches for a pattern like` "id":123 `and outputs that match using` -o'.
- `grep -o' 0-9*" Extracts only the numeric part from the above string.

🛠️ Our Complete Testing Suite

Web Interface (index.html)

- Visual feedback with JavaScript
- Interactive testing
- Great for learning
- Real-time results

Command Line JavaScript (test_runner.js)

(Optioanl) Run this when you know `node.js` and how to run a `node.js` script.

- Node.js automation
- Detailed JSON validation
- Programming examples

CURL Scripts (test_runner_curl.sh/.bat)

- Universal tool
- Shell scripting examples
- Cross-platform support

🚀 Running the Tests

Step 1: Start API Server

```
cd api
php -S localhost:8000
```

Step 2A: Web Interface

Open `api_tests/javascript/index.html` in browser (as a file)

Step 2B: JavaScript CLI

(Optional) Run this when you know how to use `node.js`.

```
node api_tests/javascript/test_runner.js --verbose
```

Step 2C: CURL Scripts

Running the script on Windows may cause some issues; in this case, use WSL2.

```
# Linux/Mac
./api_tests/curl/test_runner_curl.sh --verbose

# Windows
api_tests\curl\test_runner_curl.bat
```

🧪 Error Testing Examples

Test Invalid Data:

JavaScript

```javascript
const invalidStudent = {
    name:"  ",  // Empty name
    email: "invalid-email",  // No @
    year: "not-a-number"  // Wrong type
};

const response = await fetch('/students', {
    method: 'POST',
    body: JSON.stringify(invalidStudent)
});

// Should return 400 Bad Request
assert(response.status === 400);
```

CURL

```bash
# Test with invalid data
invalid_data='{
    "name":"  ",
    "email": "invalid-email",
    "year": "not-a-number"
}'
response=$(curl -s -w "%{http_code}" \
    -X POST \
    -H "Content-Type: application/json" \
    -d "$invalid_data" \
    http://localhost:8000/students)
http_code=$(echo "$response" | tail -n1)
if [ "$http_code" = "400" ]; then
    echo "✅ Validation works!"
fi
```

🧪 Test 1: Connection Test

Goal: Verify the server is running and responding

JavaScript

```
async function testConnection() {
    const response = await fetch(
        'http://localhost:8000'
    );

    if (response.ok) {
        console.log('✅ Server running!');
    } else {
        console.log('❌ Connection failed');
    }
}
```

CURL

```
# Basic connection test
curl -s http://localhost:8000

# With status code check
response=$(curl -s -w "%{http_code}" \
    http://localhost:8000)
http_code=$(echo "$response" | tail -n1)

if [ "$http_code" = "200" ]; then
    echo "✅ Server running!"
else
    echo "❌ Connection failed"
fi
```

✅ Test 2: GET All Students

Goal: Verify the API returns student data correctly

JavaScript

```javascript
async function testGetStudents() {
    const response = await fetch(
        'http://localhost:8000/students'
    );
    const data = await response.json();

    // Validate structure
    if (data.success &&
        Array.isArray(data.data)) {
        console.log('✅ Structure OK');
    }

    // Validate count
    if (data.count === data.data.length) {
        console.log('✅ Count matches');
    }
}
```

CURL

```bash
# Get students and validate
response=$(curl -s \
    http://localhost:8000/students)

# Check for success field
if echo "$response" | grep -q '"success":true'; then
    echo "✅ Success field found"
fi
# Check for data array
if echo "$response" | grep -q '"data":\['; then
    echo "✅ Data array found"
fi
# Check for count field
if echo "$response" | grep -q '"count":[0-9]'; then
    echo "✅ Count field found"
fi
```

📝 Test 3: POST Create Student

Goal: Create a new student and verify the data

JavaScript

```
async function testCreateStudent() {
    const newStudent = { name: 'Test Student', ... , year: 2};
    const response = await fetch(
        'http://localhost:8000/students', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(newStudent)
    });
    const result = await response.json();
    // Validate creation
    if (response.status === 201 &&
        result.data.name === newStudent.name) {
        console.log('✅ Student created!');
    }
}
```

CURL

```
new_student='{
    "name": "Test Student CURL", ..., "year": 1
}'
response=$(curl -s -w "%{http_code}" \
    -X POST \
    -H "Content-Type: application/json" \
    -d "$new_student" \
    http://localhost:8000/students)
http_code=$(echo "$response" | tail -n1)
json_data=$(echo "$response" | head -n -1)
if [ "$http_code" = "201" ]; then
    echo "✅ Student created!"
    # Extract ID for later use
    student_id=$(echo "$json_data" | \
        grep -o '"id":[0-9]*' | \
        grep -o '[0-9]*')
fi
```

🔄 Test 4: PUT Update Student

Goal: Update a student and verify changes

JavaScript

```
async function testUpdateStudent(id) {
    const updateData = {name: 'Updated Student',major: 'Data Science'};
    const response = await fetch(
        `http://localhost:8000/students/${id}`, {
        method: 'PUT',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(updateData)
    });
    const result = await response.json();
    // Validate update
    if (result.data.name === updateData.name &&
        result.data.major === updateData.major) {
        console.log('✅ Student updated!');
    }
}
```

CURL

```
update_data=' {"name": "Updated Test Student",
               "major": "Data Science"}'
response=$(curl -s \
    -X PUT \
    -H "Content-Type: application/json" \
    -d "$update_data" \
    http://localhost:8000/students/$student_id)

# Validate update
if echo "$response" | grep -q "Updated Test Student"; then
    echo "✅ Name updated!"
fi

if echo "$response" | grep -q "Data Science"; then
    echo "✅ Major updated!"
fi
```

🗑️ Test 5: DELETE Student

Goal: Delete a student and verify removal

JavaScript

```
async function testDeleteStudent(id) {
    // Delete the student
    const response = await fetch(
        `http://localhost:8000/students/${id}`, {
        method: 'DELETE'
    });
    if (response.ok) {
        // Verify deletion
        const verifyResponse = await fetch(
            `http://localhost:8000/students/${id}`
        );
        if (verifyResponse.status === 404) {
            console.log('✅ Student deleted!');
        }
    }
}
```

CURL

```
response=$(curl -s \
    -X DELETE \
    http://localhost:8000/students/$student_id)

if echo "$response" | grep -q '"success":true'; then
    # Verify deletion
    verify_response=$(curl -s -w "%{http_code}" \
        http://localhost:8000/students/$student_id)
    verify_code=$(echo "$verify_response" | tail -n1)

    if [ "$verify_code" = "404" ]; then
        echo "✅ Student deleted and verified!"
    fi
fi
```

🔧 Advanced CURL Techniques

Debugging with Verbose Output:

```
curl -v http://localhost:8000/students
```

Save Response to File:

```
curl -s http://localhost:8000/students > students.json
```

Check Response Time:

```
curl -w "Time: %{time_total}s\n" -s http://localhost:8000/students
```

Follow Redirects:

```
curl -L http://localhost:8000/students
```

Custom Headers:

```
curl -H "Authorization: Bearer token123" \
     -H "Accept: application/json" \
     http://localhost:8000/students
```

🎯 What Makes a Good API Test?

Specific 📝

- Test one thing at a time
- Clear pass/fail criteria
- Focused validation

Reliable 🔄

- Consistent results
- No random failures
- Independent of other tests

Fast ⚡

- Quick execution
- Immediate feedback
- Suitable for frequent running

Clear 💡

- Easy to understand
- Descriptive error messages
- Good documentation

Test Anatomy: The AAA Pattern

```
# ARRANGE: Prepare test data
new_student=' {"name": "Test", "email": "test@edu"}'

# ACT: Execute the API call
response=$(curl -s -X POST -H "Content-Type: application/json" \
    -d "$new_student" http://localhost:8000/students)

# ASSERT: Verify the results
if echo "$response" | grep -q '"success":true'; then
    echo "✅ Test passed!"
else
    echo "❌ Test failed!"
fi
```

Every good test follows this pattern!

📋 Testing Checklist

For Each Endpoint Test

- [ ] **HTTP Status Code** - Is it what we expect?
- [ ] **Response Structure** - Does it match our API spec?
- [ ] **Data Types** - Are fields the right type?
- [ ] **Required Fields** - Are mandatory fields present?
- [ ] **Business Logic** - Does the data make sense?
- [ ] **Error Handling** - What happens with bad input?

🚨 Common Testing Mistakes

❌ Don't Do This:

- Test multiple things in one test
- Ignore HTTP status codes
- Assume data structure without checking
- Skip error scenarios
- Make tests dependent on each other

✅ Do This Instead:

- One assertion per test concept
- Always check status codes first
- Validate response structure
- Test both success and failure cases
- Make tests independent

💻 CURL vs JavaScript: When to Use What?

Use CURL When:

- Quick manual testing
- Shell script automation
- CI/CD pipelines
- Server environments
- Learning HTTP basics
- Cross-platform compatibility

Use JavaScript When:

- Web application testing
- Complex data validation
- Interactive interfaces
- Learning programming concepts
- Frontend integration testing
- Rich error reporting

📚 Key Takeaways

Testing is Essential 🎯

- Prevents bugs in production
- Builds confidence in your code
- Improves code quality

Multiple Tools Available 🛠️

- JavaScript for web-based testing
- CURL for command-line automation
- Both achieve the same goals

Start Simple 🚀

- Connection tests first
- Basic functionality second
- Edge cases third

Be Systematic 📋

- Test all HTTP methods
- Validate response structure
- Check error scenarios